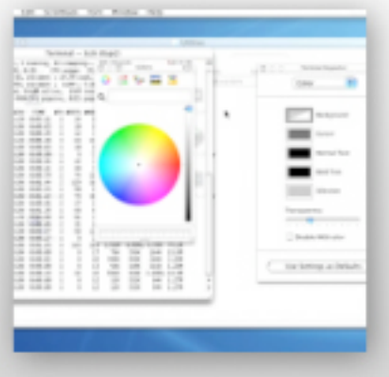
IOKit Techniques

Threading in the Darwin Kernel

Session 100



IOKit Techniques

Threading in the Darwin Kernel

Craig Keithley
I/O Technology Evangelist
keithley@apple.com

IOKit Techniques

Threading in the Darwin Kernel

Godfrey van der Linden
IOKit Architect

Introduction

- An idle Mac OS X system has over 100 threads—
  Most of these threads are ‘sleeping’
- This session will discuss and categorise these threads…

  …and how they interact with each other
- For developers writing kernel extensions (kexts) or need to be thread ’aware’

What You'll Learn

- You should get a better understanding of how threads work on Mac OS X
- How IOKit does its synchronisation
- How does Mac OS X do hot unplug in a multi-threaded environment

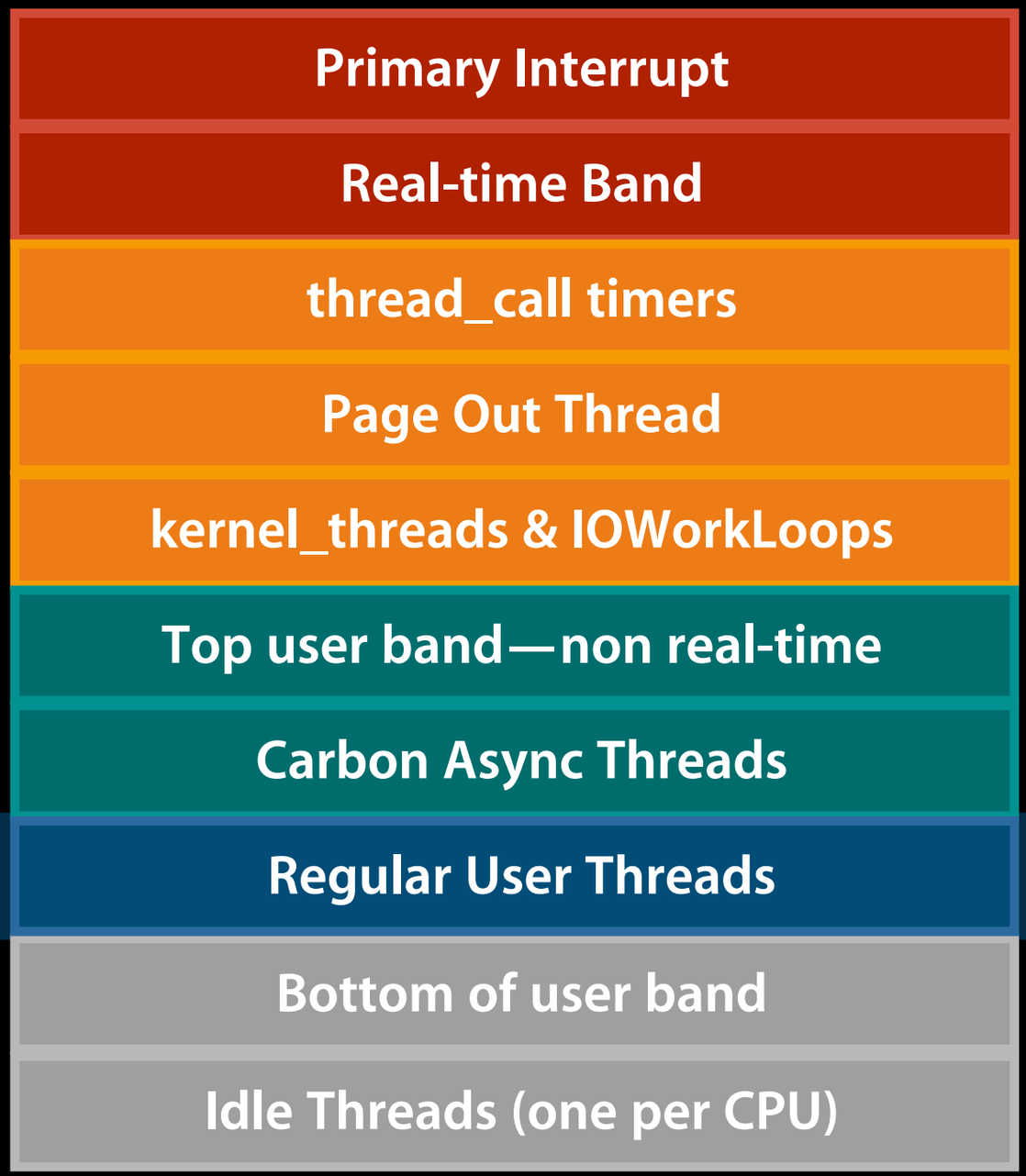
Thread Scheduling

Or how what you don't know can hurt you

- Over the next few slides I shall explain how threads on a Mac OS X system interact with each other
  - Mac OS X's thread priority bands
  - How the Mac OS X dispatcher works
  - What does a scheduler do?
  - What does high priority mean?
  - Priority inversions

X's Thread Priority Bands

"I am not a number"



How the Mac OS X Dispatcher Works

Do one thing well, then move on

- The dispatcher is part of the scheduler—It blocks a thread, then selects and runs the next thread
- Block: Wait for I/O or some other event: Thread is added to a wait queue till event occurs
- Preemption: On atom expiration the current thread is returned to the run-queues awaiting the CPU
- Every time the dispatcher is invoked it selects the highest priority **runnable** thread and runs it

What Does a Scheduler Do?

Oversight committee

- The Scheduler's job is to balance the CPU load on a system…

  …and to try to identify naughty/greedy threads
- Manipulates the run-queues
- Collecting statistics on a thread's CPU usage, computes and changes relative priorities

What Does a Scheduler Do?

Example 1: Time Share thread

If a thread is running for sufficient time then its current priority is dropped

Example 2: Misbehaving real-time thread

Say a real-time thread goes into an infinite loop—The scheduler detects this and sets the thread to time share

So What Does High Priority Mean?

You only have 100% CPU? But I want more!

- Nothing can make slow code run fast
- High priority does not make your thread run fast
- It gives you a reasonable chance of getting low latency
  - Remember you compete with other threads in your priority band
- Using too high priority may starve tasks that you depend on
  - This is called a ‘priority inversion’

How to Decide Thread Priority

- You need to really understand your latency requirements
  - **Extremely Low**
    Are you reacting to user events, for instance a midi-keyboard, or a mixing board?
  - **Who Cares?**
    Do you need to react to data off the Internet?
  - **Low**
    I'm waiting for the disks to get around to me

Priority Inversion!

I want highest priority except when I don't!

- Although priority inversions can happen anywhere; the most common cause is the real-time model
- Unlike most other OSes, Mac OS X gives its real-time band the highest priority even higher than I/O
  - This gives us extraordinarily low 'jitter'…

…but at the cost of complexity—you need to give the system time to do its job

Priority Inversion Strategies

- **Best** Lower your priority
  - Ask yourself why you need to execute at this high priority?
- **Good** Split into producer/consumer model
  - With consumer at real-time and producer lower
- **Bad** Enforce no more than about 90% CPU usage
  - Issuing a 1ms usleep() every 10ms will do this
- Don’t forget the user interface!

IOWorkLoop

The IOWorkLoop is not a thread!

- It is better to think of it as an IOWorkGate
  - Unfortunately we can't change the name
- A work loop is a container for a gate (i.e., a recursive lock), a list of event sources and a thread that is **only** used for delivering interrupt events
- The work loop's gate is closed around all event source Action routines

IOWorkLoops and Driver Stacking

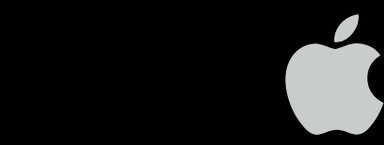
If you only have one lock you can't deadlock! Cool.

- By far the majority of IOKit drivers do not create their own work loop, but instead use their provider's work loop
  - This is a recursive statement
- In general only PCI and motherboard device drivers create work loops and they **must** override the IOService::getWorkLoop() function
- Use the system's default work loop, if you don't have very tight timing **and** you don't expect a lot of interrupts

IOEventSource

The Action routine is synchronous w.r.t. the work loop

- All event sources have an action routine, an owner and are usually registered on a work loop
- An Action routine is a call-out function that is called when the event occurs
- All Action routines are called synchronously w.r.t. all registered event sources on a work loop
- A driver stack tends to use the ‘ultimate’ provider’s work loop
  - See IOService::getWorkLoop()



IOFilterInterruptEventSource

What, where, who, why?

- This event source is used to deliver hardware interrupts to a driver
- It takes the interrupt and kicks the IOWorkLoop's thread
  - This is the only mechanism that unblocks the work loop's thread
- You must always implement a Filter routine for all PCI device drivers
- The Action routine is synchronous w.r.t. the work loop but the Filter is totally asynchronous

IOTimerEventSource

Is it time already?

- A timer can be used for many reasons, but probably the most common is for hardware timeouts
- The IOKit timer uses the kern/thread_call.h APIs
  - Timer threads have higher priority than interrupt threads
- **Warning** there is no synchronous way of canceling a timer
  - The safest way to delete a timer is to wait for it to expire and not re-arm it
- The timer's Action routine is synchronous w.r.t the IOWorkLoop

IOCommandGate

You want to do what?

- A command gate isn't a lock—It wraps the work loop's gate
  - I.e., For all command gates there is only one gate per work loop
- Command gates allow you to run code synchronously w.r.t. the work loop without a thread switch
- The gate is a recursive lock which is why we use the somewhat clunky runAction() API

commandSleep/commandWakeup

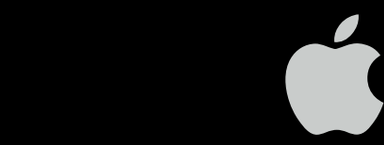
No work? It's nap time.

- The command gate's sleep and wakeup routines allow you to block a client thread
- For example: A data acquisition device driver
  - Your application asks for data and you don't have any data—Call commandSleep() to block
  - Some data arrives, use your interrupt's Action to commandWakeup() your application
- Used to emulate interrupt callouts

Interactions With BSD Locking

So what is a funnel then?

- It doesn't matter to IOKit Developers—Apple writes bridging code between BSD & IOKit locking
- BSD uses funnels to emulate the 'single' threaded kernel that BSD code assumes—we have generalised this a little into a network lock and an everything else lock
- Writing code that switches between funnels is difficult and definitely not for the faint hearted
- Funnels can cause long delays on work loops
  - Subtle priority inversion



Synchronous Device Tear Down

"Oh No! My device has gone!"

- Tearing down a stack was one of the hardest things to get right in IOKit
- The difficulty is that it is very easy to delete a driver object while a client thread is calling down the stack

Synchronous Device Tear Down

"Oh No! My device has gone!"

- Tearing down a stack was one of the hardest things to get right in IOKit
- The difficulty is that it is very easy to delete a driver object while a client thread is calling down the stack

Synchronous Device Tear Down

"Oh No! My device has gone!"

- Tearing down a stack was one of the hardest things to get right in IOKit
- The difficulty is that it is very easy to delete a driver object while a client thread is calling down the stack

Synchronous Device Tear Down

- Our solution has been to use the work loop stacking to provide a synchronising context
- When a bus discovers a missing device, it calls terminates while on the work loop, on the device nub that has gone, i.e., nub->terminate()
- IOService::terminate() calls doTerminate()

```
doTerminate() {
    client->willTerminate();
    for each client
        client->doTerminate();
    client->didTerminate();
}
```

Synchronous Device Tear Down

- The responsibility of a driver depends on its position in the stack—If it is an intermediate driver it will probably override willTerminate()
- willTerminate() will complete all currently outstanding I/Os with a kIOReturnOffline error
- By the time we get to the top of the driver stack **all** outstanding I/Os should be completed with an error

Synchronous Device Tear Down

- If the driver is on the top of the stack it is supposed to override didTerminate()
- When a driver receives the didTerminate() call it :-
  - Stops all future calls to the provider
  - Waits, asynchronously, for all provider calls to return
  - Close()s the provider as soon as possible
- This can be tricky to implement well
  - Usually IOKit provides the TOS drivers, but a user client is TOS

Wrap Up

Roadmap

102 Open Source at Apple An Overview of Apple's Open Source Efforts	**The Mission** **Tue, 10:30am**
107 Kernel Programming Interfaces Preview of Upcoming APIs for KEXTs	**The Marina** **Wed, 10:30am**
111 Writing Treaded Applications on Mac OS X	**Pacific Heights** **Thu, 2:00pm**
501 Bluetooth Update Bluetooth in Mac OS X	**The Marina** **Tue, 9:00am**

Roadmap

503 USB Update USB in Mac OS X	**The Marina** **Tue, 2:00pm**
505 FireWire in Depth	**The Marina** **Tue, 5:00pm**
FF001 Darwin	**North Beach** **Tue, 3:30pm**
FF006 FireWire and USB	**North Beach** **Wed, 3:30pm**

Who to Contact

Craig Keithley
I/O Technology Evangelist
keithley@apple.com

http://developer.apple.com/wwdc2003/urls.html

For More Information

Reference Library

Documentation

I/O Kit Fundamentals

ADC Home > Documentation > Device Drivers & I/O Kit > I/O Kit Fundamentals

Kernel Extensions Tutorials

ADC Home > Documentation > Device Drivers & I/O Kit > Kernel Extensions Tutorials

Writing an I/O Kit Device Driver

ADC Home > Documentation > Device Drivers & I/O Kit > Writing an I/O Kit Device Driver

Reference Library

Documentation (Cont.)

Kernel Programming
ADC Home > Documentation > Darwin & Core OS > Kernel Programming

Device Drivers (Kernel-I/O Kit) API Reference
ADC Home > Documentation > Device Drivers & I/O Kit > Device Drivers (Kernel-I/O Kit)

Kernel Extensions (Kernel/libkern) API Reference
ADC Home > Documentation > Device Drivers & I/O Kit > Kernel Extensions (Kernel/libkern)

Handling Events
http://developer.apple.com/techpubs/macosx/Darwin/General/IOKitFundamentals/HandlingEvents/chapter_8_section_1.html

Reference Library

Technical Notes

TN2028: Threading Architectures
http://developer.apple.com/technotes/tn/tn2028.html

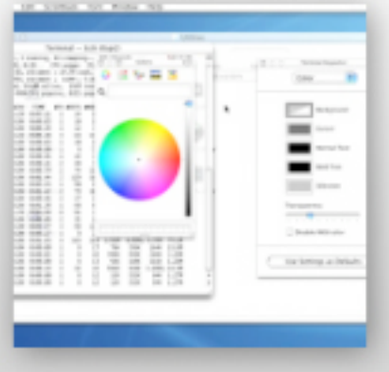
Source Code

Darwin Kernel Sources
http://developer.apple.com/darwin

IOWorkLoop & IOEventSources
xnu/iokit/Kernel

Simple User Client
ADC Home > Sample Code > Devices and Hardware > IOKit > SimpleUserClient

Q&A



Craig Keithley
I/O Technology Evangelist
keithley@apple.com

http://developer.apple.com/wwdc2003/urls.html

WWDC 2003

Worldwide Developers Conference